**C# 9 – 5 new features Read in 10 mins**

[C#](https://stackify.com/what-is-c-used-for/) is rapidly approaching its third decade of life. Originally released in 2000, the language has grown and changed across 9 major versions. Once a knock off of Java in all but name has jumped out ahead on many aspects.

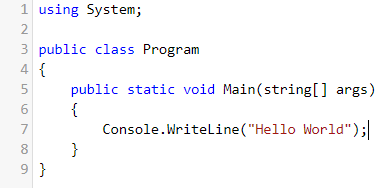
Throughout its life, the release of a new version of the language has been highly coupled with releases of new versions of Visual Studio as well as with releases of the .NET runtime. This coupling has actually reduced the pace of innovation.

## Proposed C# 9 Features

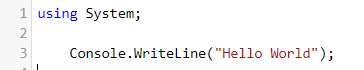
## 1. Top-level statements

## *****Top-level statements***** remove unnecessary ceremony from many applications.

Writing a simple program in C# requires a remarkable amount of boilerplate code. Consider the canonical "Hello World!" program.

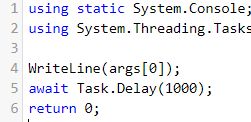


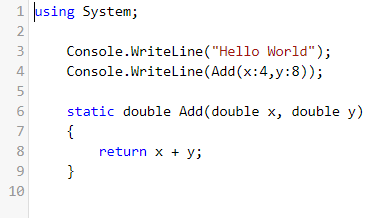
With top-level statements, you can replace all that boilerplate with using statement and the single line code that does the work:



But how it is working, because there’s no namespace, class, method and how does it even know to call this. [Top of Form](https://www.gingersoftware.com/download)Well, what C# 9 does, if you have a class file with a top level statement, then that was assumed to be the entry point of your application. So, it’s taking some of ceremony away from creating an initial console application. Top-level statements are great for small console programs and utilities. It makes life a little simpler.

The program has to occur after the using and before any type or namespace declarations in the file, and the top level statement can be present only in one file within the project, just as you can have only one Main method today. If you want to return a status code you can do that. If you want to await things you can do that. And if you want to access command line arguments, args is available as a "magic" parameter.



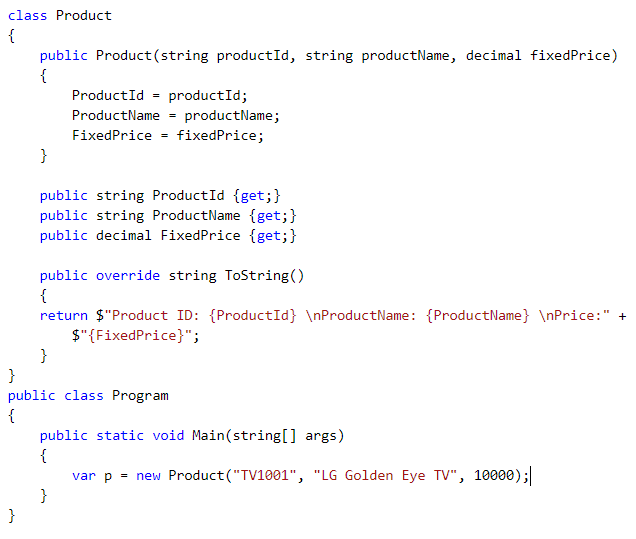
Local functions are a form of statement and are also allowed in the top level program. It is an error to call them from anywhere outside of the top level statement section.

## 2. Init only setters

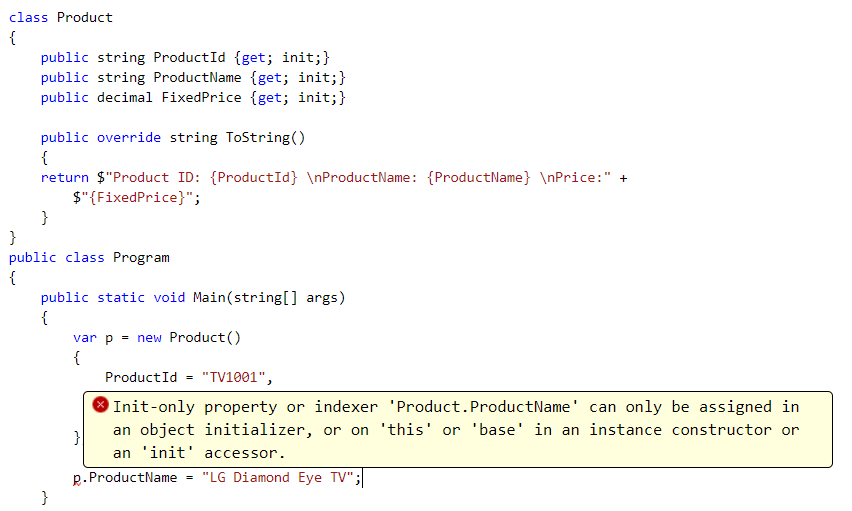
## Product as a model, now if you see this Product type. Now in this type we have three properties ProductId, ProductName and FixedPrice. And these three properties have both getter and setter accessors. The problem here is the object which will be created out of Product is mutable. Meaning, after the object is created, we can go ahead and keep changing the values of this properties like this

## C:\Users\gurum\OneDrive\Desktop\mukesh\Blog\InitOnly-1.PNG

But if we want to make the properties **immutable** means, once the property has been initialized, once again we should not allow the users to modify the values of the property. We can do that by using constructor and pass through the constructor we can make the property as immutable.



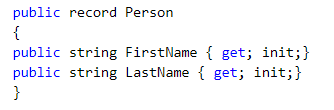
But, as you can see there is so much of boilerplate code that needed to support it. Now, here comes the new **C# 9** feature **Init only setter**. Instead of using **Set** accessors, we can use **init** keyword. It will make the properties as **immutable** or **read-only**. If we try to change the value of the property once property is initialized, it won’t allow, compiler will throw the error. And this is the big change, it is not so much of a new feature, but it is an extremely convenient feature which can get rid of lot of boilerplate code.



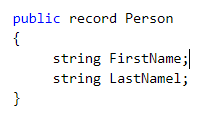
**3. Records**

Records are used to define an immutable reference type and behave like a value type. That is once the instance of the record has been defined then we can’t modify the values of any properties of the **record** instances.

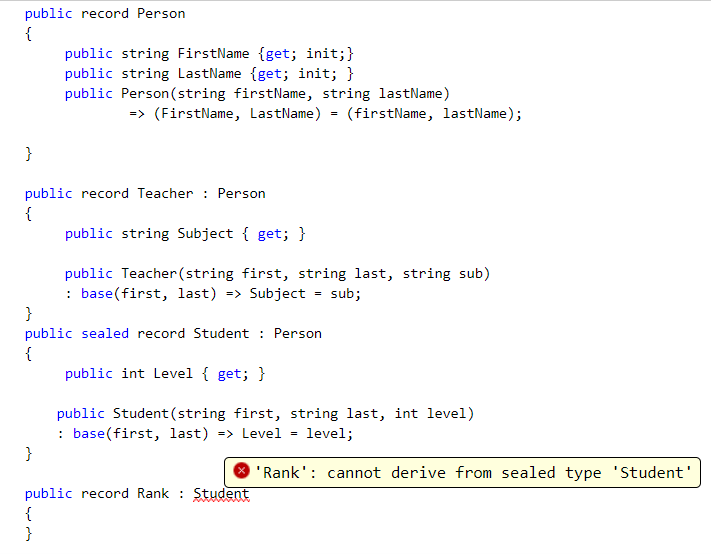
To make the whole object immutable you have to set **init** keyword on each property if you are using an implicit parameterless constructor:



The important thing to know here with **Records**is that members are implicitly **public** if you don’t precise it. Then the following class declaration is similar to the previous above and **Records** introduce also **public init-only auto-property** (if you don’t to use explicitly private fields) which is a shorthand of the previous declaration (same meaning):



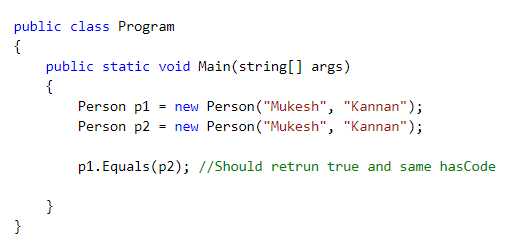
Records support **inheritance**. You can declare a new record derived from Person as follows and you can also **seal** records to prevent further derivation:



## Value-based equality

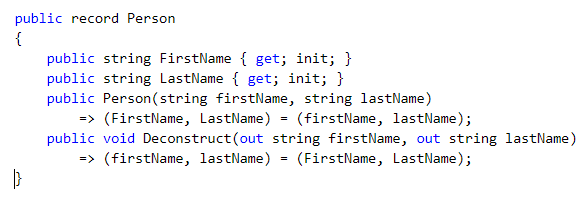
By default, regular classes are considered equal when they share the same underlying *reference*. This means that, even if you create two instances of the same class, holding the exact same values, they will not be considered equal.

This behaviour changes when you declare your class as a record. Rather than comparing the object’s reference, records are compared by value. This means that two different objects holding the same values will be considered equal and hashCode will be same. Records are defined not by their identity, but their contents.



## [Positional records](https://devblogs.microsoft.com/dotnet/c-9-0-on-the-record/" \l "positional-records)

[Sometimes it’s useful to have a more positional approach to a record, where its contents are given via constructor arguments, and can be extracted with positional deconstruction. It’s perfectly possible to specify your own constructor and deconstructor in a record:](https://devblogs.microsoft.com/dotnet/c-9-0-on-the-record/" \l "positional-records)



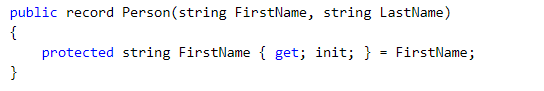
But there’s a much shorter syntax for expressing exactly the same thing (modulo casing of parameter names):

C:\Users\gurum\OneDrive\Desktop\mukesh\Blog\Records-6.PNG

This declares the public init-only auto-properties *and* the constructor *and* the deconstructor, so that you can write:

C:\Users\gurum\OneDrive\Desktop\mukesh\Blog\Records-7.PNG

If you don’t like the generated auto-property you can define your own property of the same name instead, and the generated constructor and deconstructor will just use that one. In this case, the parameter is in scope for you to use for initialization. Say, for instance, that you’d rather have the FirstName be a protected property:

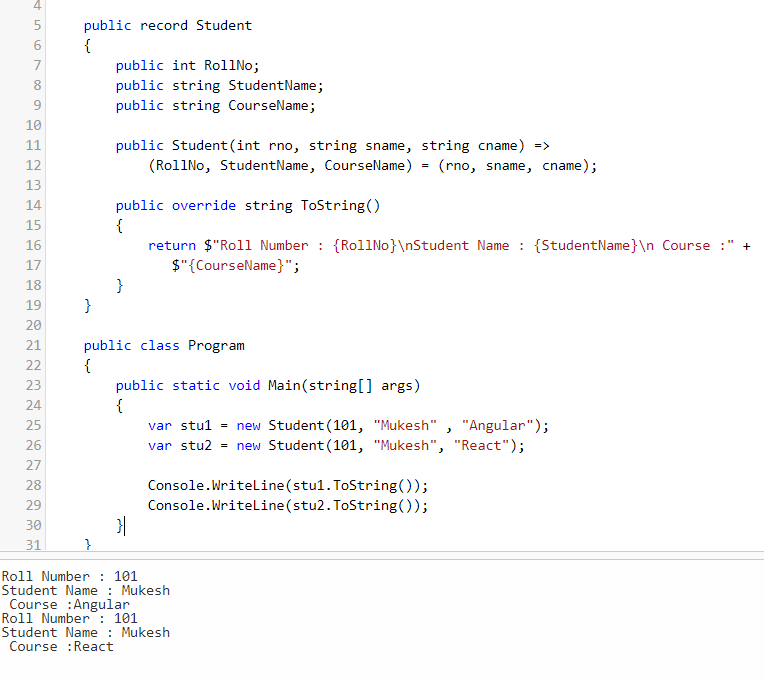


**A positional record can call a base constructor like this:**

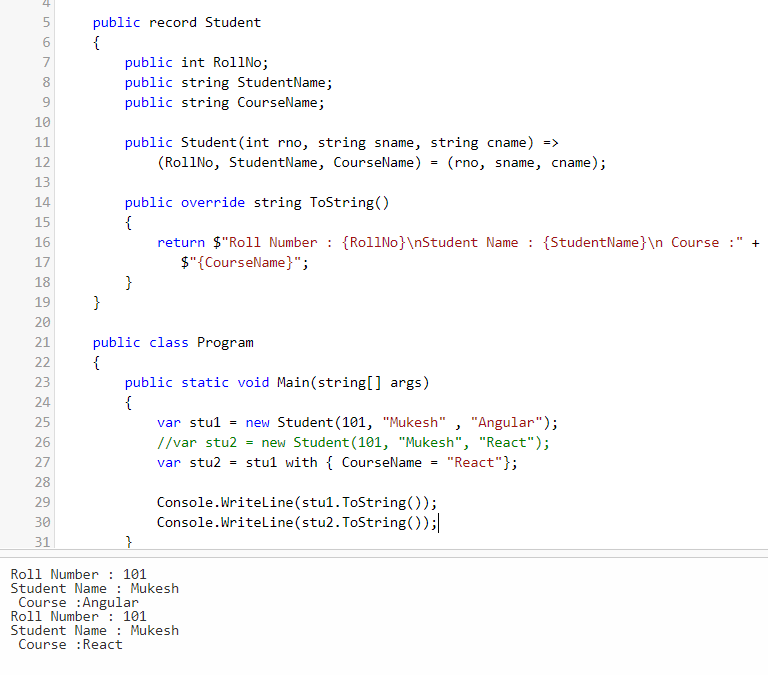
C:\Users\gurum\OneDrive\Desktop\mukesh\Blog\Records-9.PNG

# **With Expression**

Just remember that, when we are working with immutable data, the most common task, we perform will be, creating a new values from an existing one’s to represent the new state. And this technique is popularly called as an non - destructive mutation. Now, let us understand some important points about the **‘With Expressions’**. The use object initializer syntax to state, what is different in the new object from old object. We can set multiple properties using the **With Expression**. The ‘**with expression’** causes the copy constructor to get called and then applies the object initializer on the top to change the properties accordingly. Let us observe the feature,



As we expected we got the output. Now, everything is working fine. But, assume that if the student record consist of 40 properties. Using the above method will not be advisable. Because, it will require lots of boilerplate code. In previous version, by using copy constructor we can create a new object using the old object and then we shall update the new object property as per the requirement. Now, that can be handle in much more efficient way using our new **C# 9** feature **‘with expression’** by following way and we will get the same result.



Hope you have understand the usage of ‘With Expression’.

**4. Improved pattern matching**

**Pattern matching** has been introduced in **C# 6** and has well evolved since. The latest improvement was pretty interesting on **C#** which has been released last year. Pattern matching allows the developer to match a value (or an object) against some patterns to select a branch/block of the code.

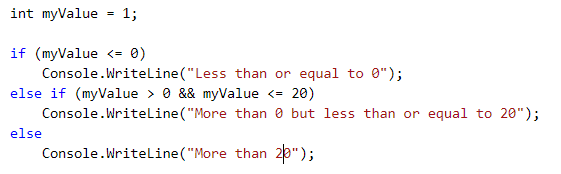
**Relational patterns**

**C# 9** allows you to use relational pattern which enables the use of **<, >, <=** and **>=** in patterns like this:

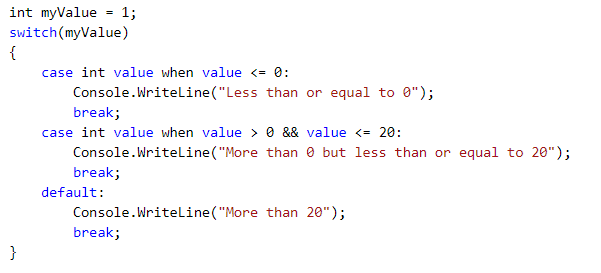
**What We Currently Have**

So first, let’s mock up a problem in **C# 8** that we should be able to solve with **Relational Pattern Matching** in **C# 9** to actually be able to see the power.

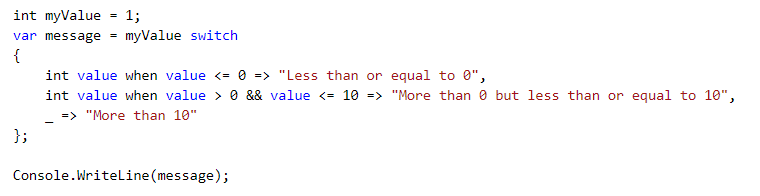
Let’s take the following code :



How could we improve it? Well, we can still use If/Else from above in most cases just for it’s simplicity. But, we could use **Pattern Matching** which is introduced in **C# 7** with a switch statement. It’ll looks like this:

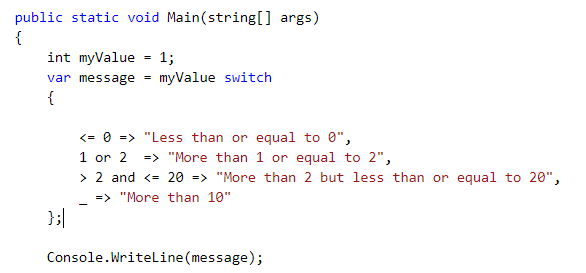


Not bad, right? But we don’t really add anything to If/Else statements, in fact we actually added even more code here. If we are using **C# 8**, then we can use **Switch Expressions**. That would be like this:



**Logical patterns && Relational Pattern Matching In Switch Statements/Expressions**

**C# 9** lets you use logical operators like **‘and’**, **‘or’** and **‘not’**, they can even be combined with relational patterns like this and We can still improve above code like this:



# **5. Target-typed ‘new’ expressions**

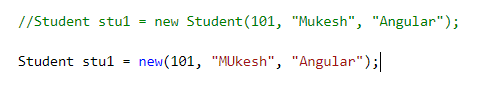
We can omit the type in the ‘new expression’ when the created object is already known.

C:\Users\gurum\OneDrive\Desktop\mukesh\Blog\new expression-1New.PNG

Output:

C:\Users\gurum\OneDrive\Desktop\mukesh\Blog\new expression-1Newoutput.PNG

We know that the above syntax is available from the earlier version of C#. **new** expressions in C# have always required a type to be specified (except for implicitly typed array expressions).In C#-9, we don’t have to mention the record type on class name for the new expression. Let update the above code like below and we will get the same result.



The most common use of this feature is, while defining the field declarations. This is particularly nice when you have a lot of repetition, such as in an array or object initializer:

C:\Users\gurum\OneDrive\Desktop\mukesh\Blog\new expression-3.PNG